経営情報システム学特論１
５．検証（１）

SS専攻 経営情報システム学講座 客員
石川 冬樹
f-ishikawa@nii.ac.jp

目次

- 性質の検証
- デッドロック
- 安全性
- 活性（進行性）

性質

- 検証における性質（property）：プログラムやそのモデルに対し，起きうるすべての実行パスにて成り立つべき命題

  すなわち検証において「正しさ」の
  基準となり確認する対象

- 逐次プログラムの場合，

  実行後の結果が期待された条件を満たすこと
  プログラムが停止すること

（初回より）「正当性」・「検証」

V&V
検証（Verification）
妥当性確認（Validation）

- **（正当性）検証**：「成果物を正しく作っている？」
  その成果物が満たすべき性質（基準）を満たす？

こちらを扱う！

- **妥当性確認**：「妥当な成果物を作っている？」
  自身や顧客の本来の要求に合致している？

（初回より） モデル検査にて扱う性質

Reachability *到達可能性*	ある条件の下ある状況に到達しうる 「初期画面に戻る操作の列が常にある」
Safety *安全性*	ある条件の下ある状況に到達することがない 「踏切が空いた状態で電車が通過することはない」
Liveness *活性*	ある条件の下ある状況にいつか必ず到達する 「登録をするといつか必ず確認メールが届く」
Deadlock-freeness	デッドロックが起きない
Fairness *公平性*	ある条件の下ある状況が無限回起きる（ない） 「ユーザが無数にリクエストを送れば無数に返事が来る（他のユーザ等にブロックされ続けることはない）」

LTSAにおける検証手段

- 正しい動作を定めるプロセスを与え，それに合致しない場合は誤りであると考える
  - propertyキーワード
- キーワードを用いて特定の性質を表現する
  - （既出）ERRORプロセス（図の上での状態「-1」）
  - progressキーワード（進行性を表現）
- 成り立つべき性質を直接論理式として与える
  - assertキーワード（次回）

目次

- 性質の検証
- デッドロック
- 安全性
- 活性（進行性）

デッドロック

- デッドロック： すべての構成プロセスがブロックされている状態
  - 並行システムにおける典型的なバグ
    （不適切な設計・実装により発生する問題）
  - 初回資料でも言及（食事する哲学者）
  - LTSAで図示された状態遷移上で理解する場合，出ていく辺のない状態に相当
  - LTSAで自動的に発見させる場合，Safety checkにて発見可能

最も基本的な例題

- プリンターとスキャナーを両方確保してできる仕事をする2つのプロセス

```
PRINTER = ( pget -> print ->
          pput -> PRINTER ).
SCANNER = ( sget -> scan ->
          sput -> SCANNER ).

P = ( pget -> sget -> scan ->
     print -> pput -> sput -> P ).
Q = ( sget -> pget -> scan ->
     print -> sput -> pput -> Q ).

||SYSTEM = ( p:P || q:Q || {p,q}::PRINTER
           || {p,q}::SCANNER ).
```

デッドロック発生の原因

- デッドロックの必要十分条件 [Coffman et al, 1971]
  - Serially reusable resources：
    複数プロセスが相互排他制御を伴う資源を共有する
  - Incremental acquisition：
    各プロセスは，他の資源を待ちつつ，自身がすでに確保した資源を保持し続ける
  - No preemption：
    一度確保された資源は，プロセスにより自発的にのみ（強制的ではなく）解放される
  - Wait-for cycle：
    各プロセスが，その後方プロセスが待つ資源を保持しているようなプロセスのサイクルが存在する

デッドロック回避（１）

- Serially reusable resources：
  複数プロセスが相互排他制御を伴う資源を共有する
- アプリケーション上の要求であれば避けられない
- Incremental acquisition：
  各プロセスは，他の資源を待ちつつ，自身がすでに確保した資源を保持し続ける
- 複数資源の確保が必要であれば保持するのは当然で，各プロセスが必要な資源をすべて一括で確保するようにすればこの状況は避けられる
  （例：「aとb両方に対する1つのロックを設ける」）
  ただし資源が多い際には（賢いスケジューリングでもしない限り）非効率となり得る

最も基本的な例題（改良案１）

- ロックの同時取得版

```
PRINTER = ( ps.get -> print -> ps.put -> PRINTER ).
SCANNER = ( ps.get -> scan -> ps.put -> SCANNER ).

P = ( ps.get -> scan -> print -> ps.put -> P ).
Q = ( ps.get -> scan -> print -> ps.put -> Q ).

||SYSTEM = ( p:P || q:Q || {p,q}::PRINTER || {p,q}::SCANNER ).
```

デッドロック回避（２）

- No preemption：

  一度確保された資源は，プロセスにより自発的にのみ（強制的ではなく）解放される

- 監視機構を設けて強制解放することは考えられるが，うまくやらないとあるプロセスがいつまでも進まない問題（ライブロックあるいは「不公平」）が発生することがある（安全のため採用する手はあり）

- Wait-for cycle：

  各プロセスが，その後方プロセスが待つ資源を保持しているようなプロセスのサイクルが存在する

- 資源を確保する順序を一定にすることにより避けられる（大原則として適用することが多い）

最も基本的な例題（改良案２）

- 自主解放版

```
PRINTER = ( pget -> PRINTERUSE ),
PRINTERUSE = ( print -> PRINTERUSE
             | pput -> PRINTER).
...

P = ( pget ->
     ( sget -> scan ->
        print -> pput -> sput -> P
      | timeout -> pput -> P
     )
   ).
...
```

無限にあきらめ続ける可能性もあるが・・・
LTSAではprogress checkも可能（後述）

最も基本的な例題（改良案２′）

- 強制解放版（ほぼ同様）

```
PRINTERMONITOR = ( pget ->
                 ( pput -> PRINTERMONITOR
                 | timeout ->
                   ( pyield ->
                     pput -> PRINTERMONITOR
                   | pput -> PRINTERMONITOR
                   )
                 )
               ).
P = ( pget ->
    ( sget -> scan ->
        print -> pput -> sput -> P
    | pyield -> pput -> P
    )
  ).
```

最も基本的な例題（改良案３）

- 同じ順序で確保

```
PRINTER = ( pget -> print ->
          pput -> PRINTER ).
SCANNER = ( sget -> scan ->
          sput -> SCANNER ).

P = ( pget -> sget -> scan ->
     print -> pput -> sput -> P ).
Q = ( pget -> sget -> scan ->
     print -> sput -> pput -> Q ).

||SYSTEM = ( p:P || q:Q || {p,q}::PRINTER
           || {p,q}::SCANNER ).
```

モニターのネスト

- 資源確保（ロック）を含めたモニターの操作に関し，「順次確保して一括利用」というよりも「ネストによる順次確保」もよくある
  - 「Aを使っていくつかタスクをするが，その際の一部ではBも必要」といったときに，「Bを解放する・生産する」という側がAも必要とするとデッドロック
  - Javaでsynchronizedを使ってロックを確保しつつ，その中でさらにsynchronizedやwait，それらが含まれたライブラリメソッドを使うと発生する（前回の例）

前回の例

historyを複数スレッドから同時に更新しないよう
synchronizedしておこう

```
synchronized protected boolean produce() {
  sem.release();
  history.add(sem.availablePermits());
  return true;
}

synchronized protected boolean consume() {
  try {
   sem.acquire();
  } catch (InterruptedException e) { ... }
  history.add(sem.availablePermits());
  return true;
}
```

前回の例（LTSAイメージ）

```
synchronized protected boolean produce() {
  sem.release();
  history.add(sem.availablePermits());
  return true;
}

synchronized protected boolean consume() {
  try {
   sem.acquire();
  } catch (InterruptedException e) { .
  history.add(sem.availablePermits());
  return true;
}
```

P = (
lock ->
s.put ->
work ->
unlock -> P).

C = (
lock ->
s.read[1..N] ->
s.get ->
work ->
unlock -> C)

目次

- 性質の検証
- デッドロック
- 安全性
- 活性（進行性）

安全性

- 安全性（Safety）：望ましくない状況に到達しないこと
  - xの値が負になることはない
  - プリンターのロックを同時に2つ以上のプロセスが確保していることはない
  - 踏切が空いた状態で電車が通過することはない
  - ．．．
- 状態をすべて探索し，各状態を調べればよい
  - LTSAではデッドロック検査は，検査方法が同様なのでSafety Checkの一部となっているが，分類としては区別されることもある

安全性検証：例 1

- クライアントは，コマンドを送ったら，返事が来るまでコマンドは送ってはならない

```
CLIENT1 = (
  process -> command -> respond -> CLIENT1
) @{command,respond}.
```

こちらは正しい

これらの遷移以外は
内部動作として無視

```
CLIENT2 = (
  process -> command -> CLIENT2
  | process -> command -> respond -> CLIENT2
) @{command,respond}.
```

こちらは正しくない

安全性検証：例１

- 方法１：同期して動作し許されない動作が起きた際にはERRORに至るような監視者プロセスを書いて合成する

```
CLIENTCHECKER = C1,
C1 = (
  command -> C2
  | respond -> ERROR
),
C2 = (
  respond -> C1
  | command -> ERROR
).
```

安全性検証：例１

- 方法２：正しい振る舞いを表現するプロセスをpropertyキーワードで与え合成する
  - 合成する必要がある点に注意
  - 裏で起きることは方法１と同じ：
    ツールが生成する状態遷移を見てみよ

```
property GOODCLIENT =
  ( command -> respond -> GOODCLIENT ).
```

安全性検証：例２（先週の例）

- 0以上N以下の値しかとるべきでない資源変数
  - 変数値の更新（資源の生成・消費）側が留意

```
const N = 2
RESOURCE = RESOURCE[0],
RESOURCE[u:0..N] = (
  put -> RESOURCE[u+1]
  | get -> RESOURCE[u-1]
),
RESOURCE[-1] = ERROR,
RESOURCE[N+1] = ERROR.
PRODUCER = ( put -> PRODUCER ) +{get,put}.
CONSUMER = ( get -> CONSUMER ) +{get,put}.
||SYSTEM = ( {p1,c1,c2}::RESOURCE || p1:PRODUCER
                    || c1:CONSUMER || c2:CONSUMER ).
```

不適切な状態を明示的にERRORにより指定

安全性検証：例２（実質は復習）

- ERRORを用いずpropertyを用いる場合

```
...
RESOURCE[-1] = ( reset -> RESOURCE[0] ),
RESOURCE[N+1] = ( reset -> RESOURCE[0] ).
...
||SYSTEM = ...
  || {p1,c1,c2}::SAFERESOURCE ).

property SAFERESOURCE = SAFERESOURCE[0],
SAFERESOURCE[u:0..N] = (
  when (u<N) put -> SAFERESOURCE[u+1]
  | when (u>0) get -> SAFERESOURCE[u-1]
).
```

挙動確認のためこちらの
ERRORは外し動き続けるようにする

propertyで定義したプロセスも合成

許される動作を定義

安全性検証：例２（実質は復習）

- この生産者と消費者の修正は先週の通り
  - まず，現在の値を確認してからget/putをするようにしなければならない
  - その「状況確認の後get/put」は原子的となるようにロックをとるようにしなければならない
  - ロックを持ったまま状況変化待ちでブロックすることを避けるため，動作できないときにはロックを手放す必要がある

目次

- 性質の検証
- デッドロック
- 安全性
- 活性（進行性）

活性（進行性）

- 活性（進行性）（Liveness/Progress）：望ましい状況にいつか到達すること
  - xの値がいつか0になる
  - 登録をするといつか必ず確認メールが届く
  - どのプロセスもプリンターのロックをいつか確保することができる
  - 有効な行動１，２，３のいずれかを常にいつか行うことができる（どれも行えないままとなることはない）
  - ．．．

「無限にいつか起きる（ブロックされ続けることがない）」という言い回しを用いることが多い

活性（進行性）検証：例１

- コインを振ったとき常に表も裏も何度でもいつか出る（どちらかが出なくなることはない）

```
COINTOSS1 = ( pick -> COIN ),
COIN = (
  toss ->
  ( heads -> COIN | tails -> COIN )
).

progress HEADSPROGRESS = {heads}
progress TAILSPROGRESS = {tails}
progress TOSSPROGRESS = {heads,tails}
```

こちらは正しい

headsが無限に起きるか？

tailsが（同上）

headsかtailsのいずれかが（同上）

活性（進行性）検証：例１

- コインを振ったとき常に表も裏も何度でもいつか出る（どちらかが出なくなることはない）

```
COINTOSS2 = (
  pick -> COIN
  | cheatpick -> CHEATCOIN
),
COIN = (
  toss ->
  ( heads -> COIN | tails -> COIN )
),
CHEATCOIN = ( toss -> heads -> CHEATCOIN ).

progress HEADSPROGRESS = {heads}
progress TAILSPROGRESS = {tails}
progress TOSSPROGRESS = {heads,tails}
```

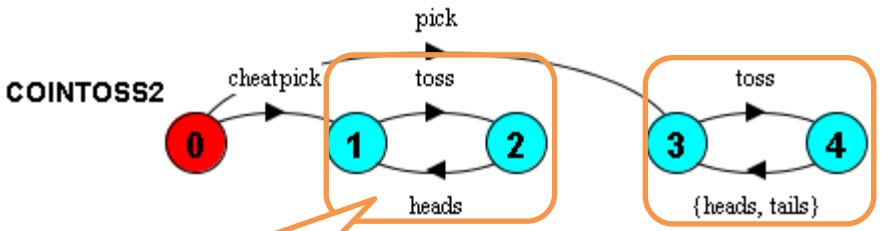
表しか出ないコインにすり替わることがある

tailsのみ違反となる

活性（進行性）検証

- LTSAの内部でやっていること
  - 遷移は無限に繰り返すようになっている
  - サイクルを検出する
  - サイクルの中に含まれている遷移（ラベル）の集合（terminal set）に該当ラベルがあるかチェックする

COINTOSS2の場合



このサイクルではterminal setが{toss,heads,tails}

このサイクルではterminal setが{toss,heads}

活性（進行性）検証

- LTSAの考え方
  - サイクルの中の遷移は特に見ず，terminal setの中に入っているかどうかだけを見ている
  - 「起きない可能性がある」ことを検出しているわけではない（普通のコインも「たまたま」無限に表だけ出続ける可能性はゼロではない）
  - 「決して起きえない」ような無限ループがある場合のみ，問題として検出している
  - つまり，「すべての非決定的な選択肢がどれも『公平に』無限回選ばれるという仮定の下で，無限にいつか起きる」かどうかを検査している

活性（進行性）検証：例 1

- 表が常に出るシナリオを明示的に試す
  - 合成に対して（のみ）優先度記述ができる

```
COINTOSS1 = ( pick -> COIN ),
COIN = (
  toss ->
  ( heads -> COIN | tails -> COIN )
).

||BROKENCOINTOSS1 = COINTOSS1 << {heads}.

progress HEADSPROGRESS = {heads}
progress TAILSPROGRESS = {tails}
progress TOSSPROGRESS = {heads,tails}
```

他遷移（τ 含め）より高優先

tailsに関し違反となる

活性（進行性）検証：例 1

- 表が常に出ないシナリオを明示的に試す
  - 合成に対して（のみ）優先度記述ができる

```
COINTOSS1 = ( pick -> COIN ),
COIN = (
  toss ->
  ( heads -> COIN | tails -> COIN )
).

||BROKENCOINTOSS1 = COINTOSS1 >> {heads}.

progress HEADSPROGRESS = {heads}
progress TAILSPROGRESS = {tails}
progress TOSSPROGRESS = {heads,tails}
```

他遷移（τ 含め）より低優先

headsに関し違反となる

活性（進行性）検証：例２

- クライアント・サーバーの例

```
CLIENT1 = ( request -> serve -> CLIENT1 ).
CLIENT2 = ( request -> serve -> CLIENT2 ).

SERVER = ( request -> serve -> SERVER ).

||SYSTEM = ( c1:CLIENT1 || c2:CLIENT2
          || {c1,c2}::SERVER )
             <<{c1.request}.

progress PROGRESS1 = {c1.serve}
progress PROGRESS2 = {c2.serve}
```

c2に関し違反となる

補足：公平性

- 公平性
  - LTSAのprogressのように，「ブロックされ続ける状況に陥ることなく，無限回いつか起きることができる」は，概念的には公平性とも見なすことができる
  - 詳しくはまた後日

まとめ

- デッドロック
  - 並行システムにおける典型的な不具合である
  - 資源確保順序の統一など原則を理解する必要がある
  - Javaではsynchronizedキーワードやライブラリ利用などによる「手軽な」ロック確保において，実際に起きている挙動について理解する必要がある
- 性質検証
  - 成り立つべき性質を明記し，検証する
  - 安全性と活性（進行性）は，性質の種類のうち特に重要なものであり，LTSAでは手軽な検証手段を用意している
- 次回： 性質の論理式による記述を扱う